# Operating Systems (2INC0)

## Action Synchronization (06)
## Checking invariants

**Dr. Geoffrey Nelissen**

**Courtesy of Prof. Dr. Johan Lukkien and Dr. Tanir Ozcelebi**

Interconnected Resource-aware Intelligent Systems

Technische Universiteit **Eindhoven** University of Technology

Where innovation starts

# Synchronization

- **Synchronization** is about **limitation of possible** program **traces** by coordinating execution such that
  - a certain invariant is satisfied
    - i.e., avoid the traces that violate the invariant
  - or the execution has some desired property
    - e.g. a certain assertion holds at a certain control point

- Typically, by blocking execution until an assertion has become true.

# Action synchronization

- relies on action counting and invariants on the counting.

- An **invariant** ***I*** is an assertion that holds at *all* control points.
  - (Example) ***I* :** "mutual exclusion is maintained"
  - (Example) ***I* :** $y \leq x$ in the program below

    …assuming <atomic> assignments…

Initially: $x=0 \wedge y=0$

**while** *true* **do** $<x := x+1>; <y := y+1>$ **od**

$\|$

**while** *true* **do** $<y := y-1>; <x := x-1>$ **od**

# Terminology: naming and counting

## Naming of actions

Initially: $x=0 \wedge y=0$

**while** *true* **do** **A:** $<x := x+1>$; **B:** $<y := y+1>$ **od**

||

**while** *true* **do** **C:** $<y := y-1>$; **D:** $<x := x-1>$ **od**

If $A$ is an action in the program, $\underline{c}A$ denotes the number of completed executions of $A$. $\underline{c}A$ can be regarded as an auxiliary variable that is initially 0 and is incremented atomically each time $A$ is executed.

$$A \quad \rightarrow \quad <A;\ \underline{c}A := \underline{c}A+1>$$

# Topology properties

**Topology invariants**: derived directly from the program text

**Example**: two actions always occurring one after the other

Initially: $x=0 \wedge y=0$

**while** *true* **do A:** $<x := x+1>$; **B:** $<y := y+1>$ **od**

$\|$

**while** *true* **do C:** $<y := y-1>$; **D:** $<x := x-1>$ **od**

Invariants:

$I_0$: $x = \underline{c}A - \underline{c}D$

$I_1$: $y = \underline{c}B - \underline{c}C$

$I_2$: $0 \leq \underline{c}A - \underline{c}B \leq 1$

$I_3$: $0 \leq \underline{c}C - \underline{c}D \leq 1$

# Example

We can prove that ***$I: y \leq x$*** holds using topology invariants

| | |
|---|---|
| $I_0: x = \underline{c}A - \underline{c}D$ | $I_2: 0 \leq \underline{c}A - \underline{c}B \leq 1$ |
| $I_1: y = \underline{c}B - \underline{c}C$ | $I_3: 0 \leq \underline{c}C - \underline{c}D \leq 1$ |

$y \leq x \quad = ?$

$= \underline{c}B - \underline{c}C \leq \underline{c}A - \underline{c}D \quad \{ I_0: x = \underline{c}A – \underline{c}D,\ I_1: y = \underline{c}B - \underline{c}C \}$

$= true \quad \{ I_2: \underline{c}B \leq \underline{c}A, \quad I_3: -\underline{c}C \leq -\underline{c}D \}$

# Operating Systems (2INC0)

## Action Synchronization (06)
## Using semaphores

**Dr. Geoffrey Nelissen**

**Courtesy of Prof. Dr. Johan Lukkien and Dr. Tanir Ozcelebi**

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

# Synchronization conditions

- Action synchronization is specified by inequalities
  - on action counts, or…
  - …on program variables *directly related to this counting.*

- We refer to such an inequality as
  - a *synchronization condition*, or
  - a *synchronization invariant*.

# Example: Producer-consumer problem

- A producer process $P_Y$ produces information…
- …that is consumed by a consumer process $P_X$.

$$P_X = \; x := 0; \; \textbf{while } \mathit{true} \textbf{ do } \; \textbf{A}: \langle x := x+1 \rangle \; \textbf{od}$$

$$\|$$

$$P_Y = \; y := 0; \; \textbf{while } \mathit{true} \textbf{ do } \; \textbf{B}: \langle y := y+1 \rangle \; \textbf{od}$$

- Synchronize $P_X$ and $P_Y$ such that the following invariant is maintained.

$$I_0: x \leq y \qquad (= \underline{c}A \leq \underline{c}B)$$

- $I_0: x \leq y$ is desired.
  - How do we enforce that invariant?

# Recall semaphores (Dijkstra)

- Non-negative integer *s* with initial value $s_0$ and *atomic* operations *P(s)* and *V(s).*

  *P(s): < await(s>0); s := s-1 >* → block until ‘*s>0’* holds, decrement

  *V(s): < s := s+1 >* → increment

- Semaphores can be used to implement mutual exclusion

# Semaphore invariants

From the definition, we derive two semaphore properties (invariants):

$$S0\colon s \geq 0$$
$$S1\colon s = s_0 + \underline{c}V(s) - \underline{c}P(s)$$

*S0, S1*: functional properties ("safety properties"). Combining the two:

$$S2\colon \underline{c}P(s) \leq s_0 + \underline{c}V(s)$$

- **Hence, semaphores realize a <u>synchronization invariant</u> *by definition.***

In addition, we have a *progress property:*

- **Blocking is allowed only if the safety properties would be violated.**

# Solve the producer/consumer problem

$P_X =$
  $x := 0;$
  **while** *true* **do**
    **A**: $\langle x := x+1 \rangle$
  **od**

||

$P_Y =$
  $y := 0;$
  **while** *true* **do**
    **B**: $\langle y := y+1 \rangle$
  **od**

Synchronize $P_X$ and $P_Y$ such that the invariant is maintained.

$I_0 : x \leq y$

Use the program topology: $x = \underline{c}A$ and $y = \underline{c}B$
hence, $I_0$ can be rewritten: $I_0 : \underline{c}A \leq \underline{c}B$

# Solve the producer/consumer problem

Introduce semaphore *s*; let *A* be *preceded by P(s)* and *B* be *followed by V(s).*

$P_X =$
  $x := 0;$
  **while** *true* **do**
    ***P(s);*** **A**: $<x := x+1>;$
  **od**

||

$P_Y =$
  $y := 0;$
  **while** *true* **do**
    **B**: $<y := y+1>;$ ***V(s);***
  **od**

From topology:

  $I_1$: $\underline{c}A \leq \underline{c}P(s)$ and $I_2$: $\underline{c}V(s) \leq \underline{c}B$

Combine with semaphore invariant ( *S2*: $\underline{c}P(s) \leq s_0 + \underline{c}V(s)$ )

  $\underline{c}A \leq \underline{c}P(s) \leq s_0 + \underline{c}V(s) \leq s_0 + \underline{c}B$

Hence, choosing $s_0 = 0$ does the job. → $I_0$ holds.

# Action Synchronization Solution (in general)

**Given**: - collection of tasks/threads executing actions *A, B, C, D*;
- a required *synchronization condition (invariant)*

$$SYNC:\ a \cdot \underline{c}A + c \cdot \underline{c}C \leq b \cdot \underline{c}B + d \cdot \underline{c}D + e$$

for non-negative constants *a,b,c,d,e* .

---

**Solution**: introduce semaphore *s*, $s_0 = e$ and replace

$A \rightarrow P(s)^a; A$ $\quad$ $B \rightarrow B; V(s)^b$

$C \rightarrow P(s)^c; C$ $\quad$ $D \rightarrow D; V(s)^d$

Exponent = Number of times V(s) or P(s) is called

# Operating Systems (2INC0)

## Action Synchronization (06)
## POSIX implementation


**Dr. Geoffrey Nelissen**

**Courtesy of Prof. Dr. Johan Lukkien and Dr. Tanir Ozcelebi**

Interconnected
Resource-aware
Intelligent Systems

Technische Universiteit **Eindhoven** University of Technology

# Counting semaphores (POSIX 1003.1b)

- Creation and destruction
  - “name” within kernel, persistent until re-boot, like a filename
    - Posix names: for portability
  - or “unnamed” semaphores, for use in shared memory
    - shared memory between processes

```
sem_t *sem;
sem = sem_open (name, flags, mode, init_val); /* name is system-wide        */
status = sem_close (sem);                     /* semaphore still reachable */
status = sem_unlink (name);                   /* now it is removed          */

status = sem_init (sem, pshared, init_val);   /* memory space for sem must be defined,
                                                 e.g. through shm or malloc */
                                              /* pshared flag: whether multiple processes
                                                 or threads share sem of not  */

status = sem_destroy (sem);
```

# POSIX semaphore operations

```
status = sem_wait (sem);        /* P(sem) locks sem  */

status = sem_trywait (sem);     /* P(sem) again                        */
                                /* but returns error if sem == 0       */

status = sem_post (sem);        /* V(sem) */

status = sem_getvalue (sem, &val);  /* current value                                    */
                                    /* when negative: absolute value = # waiters */
```

**sem** negative value is interpreted as number of waiters (length of the waiting queue)

# Example: producer – consumer with buffer of depth 4

```
#include <stdio.h>
#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>

sem_t *s, *t;
```

```
void Producer ()
{
  int i;
  for (i=0; i<10; i++) {
    sem_wait (t); printf ("Produce "); fflush (stdout);
    sem_post (s); sleep (1);
} }

void Consumer ()
{
  int i;
  for (i=0; i<10; i++) {
    sem_wait (s); printf ("Consume "); fflush (stdout);
    sem_post (t); sleep (2);
} }
```

# (cnt'd)

```
void main ()
{
  pthread_t thread_id;

  s = sem_open ("Mysem-s", O_CREAT | O_RDWR, 0, 0);
  if (s == SEM_FAILED) { perror ("sem_open"); exit (0); }
  t = sem_open ("Mysem-t", O_CREAT | O_RDWR, 0, 4);
  if (t == SEM_FAILED) { perror ("sem_open"); exit (0); }

  pthread_create (&thread_id, NULL, Producer, NULL);
  Consumer ();
  pthread_join (thread_id, NULL);
  sem_close (s); sem_close (t);
  sem_unlink ("Mysem-s"); sem_unlink ("Mysem-t");
}
```

# Output

- Produce Consume Produce Consume Produce Produce Consume Produce Produce Consume Produce Produce Consume Produce Consume Produce Consume Consume Consume Consume

(This is one of the many possible outputs.)